**Modules**

1. Define a module type called Arith containing

a type t

a value called zero

a value called one

a function (+) : t -> t -> t

a function (\*) : t -> t -> t

a function (~-) : t -> t (note that ~- is how ocaml writes unary negation).

2. Define a module called Ints that implements the Arith module type:

module Ints : Arith = struct

...

End

3. Define a module called Floats that also implements the Arith signature. Include an additional function called (/) for dividing two floats.

Recall that you can open a module to bring all of the definitions into scope, and also that you can write let open M in e to evaluate e with all of the definitions of M in scope (this is sometimes called a "local open" because M is opened in e but not outside of the scope of the let statement).

There is a more compact syntax for a local open: M.(e) evaluates e in an environment with all of the definitions of M available. For example, the following are equivalent (extra spaces added to line things up):

List. length (List.reverse (List.map List.concat [[[1;2];[3]]]));;

List.(length ( reverse ( map concat [[[1;2];[3]]])));;

4. Which of the following expressions are valid? Keep in mind that when type checking expressions, the compiler only looks at the module type of the module, and not the module itself.

Ints.(one + one)

Ints.(1 + 1)

Floats.(one + one)

Floats.(one +. one)

Floats.(1 + 1)

Floats.(1. + 1.)

Floats.(1. +. 1.)

Ints.(1. +. 1.)

Floats.(zero / one)

Check your answers in the top level, and ensure that you can explain why each works or doesn't work.

You may have noticed that you can't do much with your Ints module. You can compute one plus one, for example, but you can't even see the answer!

utop# Ints.(one + one);;

- : Ints.t = <abstr>

The problem is that the type Ints.t is abstract: the module type doesn't tell use that Ints.t is int. This is actually a good thing in many cases: code outside of Ints can't rely on the internal implementation details of Ints, and so we are free to change it. Also, as we'll see in the next lecture, we can write generic code that works with both Ints and Floats and with any other implementation of Arith (fractions, polynomials, functions, infinite-precision real numbers...the possibilities are endless).

However, the Arith interface only has functions that return t, so once you have a value of type t, all you can do is create other values of type t.

When designing an interface with an abstract type, you will almost certainly want at least one function that returns something other than that type.

When debugging, you usually want to be able to print things. An easy way to do this is to provide a to\_string function, so that you can print the strings while debugging.

5. Add a to\_string function to the Arith interface. Notice that your code doesn't compile anymore. Fix it.

Now you can write

utop# Ints.(to\_string (one + one));;

- : string = "2"

6. It is sometimes sensible to provide a conversion from your abstract type to or from some "lowest common denominator" type. Which of the following functions would make sense as part of the Arith interface? Add the sensible ones.

to\_int : t -> int

of\_int : int -> t

to\_float : t -> float

of\_float : float -> t

Think about the impact of adding these functions on other modules you might want to implement in the future.

Sometimes you actually want to expose the type in an implementation of a module. You might like to say "the module Ints implements Arith and the type t is int," and allow external users of the Ints module to use the fact that Ints.t is int.

Luckily, OCaml lets you say exactly this. If T is a module type containing an abstract type t, then T with type t = int is a new module type that is the same as T, except that t is declared as int.

7. Define the following module type:

utop# module type ArithWithInt = Arith with type t = int;;

Compare the resulting module type with the Arith module type.

8. Change the module type of Ints to expose the implementation of Ints.t:

module Ints : Arith with type t = int = struct

...

End

9. With your new implementation, which of these expressions are valid? Test your answers.

Ints.(one + one)

Ints.(1 + 1)

Ints.(1. +. 1.)

There is a slightly different way to express type constraints, which uses with type t := ... instead of with type t = ... (the difference is the := instead of =). Instead of defining type t, the := syntax removes t and replaces it everywhere. It is occasionally useful, for example to resolve errors where a type is defined multiple times.

10. At the toplevel, compare the signatures Arith with type t = int and Arith with type t := int.